https://doi.org/10.1109/TC.2017.2714672

Tackling the Bus Turnaround Overhead in Real-Time SDRAM Controllers

Leonardo Ecco and Rolf Ernst

Abstract—Synchronous dynamic random access memories (SDRAMs) are widely employed in multi- and many-core platforms due to their high-density and low-cost. Nevertheless, their benefits come at the price of a complex two-stage access protocol, which reflects their bank-based structure and an internal level of explicitly managed caching. In scenarios in which requestors demand real-time guarantees, these features pose a predictability challenge and, in order to tackle it, several SDRAM controllers have been proposed. In this context, recent research shows that a combination of bank privatization and *open-row* policy (exploiting the caching over the boundary of a single request) represents an effective way to tackle the problem. However, such approach uncovered a new challenge: the data bus turnaround overhead. In SDRAMs, a single data bus is shared by read and write operations. Alternating read and write operations is, consequently, highly undesirable, as the data bus must remain idle during a turnaround. Therefore, in this article, we propose a SDRAM controller that reorders read and write commands, which minimizes data bus turnarounds. Moreover, we compare our approach analytically and experimentally with existing real-time SDRAM controllers both from the worst-case latency and power consumption perspectives.

Index Terms—real-time and embedded systems, memory control and access, Dynamic random access memory (DRAM)

1 Introduction and Related Work

SDRAM memories are widely employed in multi- and many-core platforms, e.g. [1] and [2], due to their high-density and low-cost. However, their benefits come at the price of a complex two-stage access protocol, which reflects their bank-based structure and an internal level of explicitly managed caching. As a consequence, the execution time of a request depends on the history of previous requests, which poses a challenge from the real-time perspective. In order to tackle such challenge, several SDRAM controllers have been proposed [3], [4], [5], [6], [7].

The classical approach to build real-time SDRAM controllers relies on a combination of bank-interleaved address mapping and *close-row* policy. The former refers to a single request accessing more than one SDRAM bank, while the latter refers to the internal level of caching being flushed between consecutive requests. Extensive work has been done on strategies to select the appropriate parameters for this approach [3], [8], [9]. More specifically, on selecting the number of banks over which a single request is distributed and how the caching can be exploited within the boundary of a single request. After the parameters are defined, bandwidth guarantees can be extracted with the approaches from [10], which makes the strategy attractive for streaming applications with well defined requirements.

From the real-time perspective, the combination of bank-interleaved address mapping and *close-row* policy is very effective in scenarios in which the SDRAM data bus is narrow and/or the SDRAM requests have a large granularity. However, if that is not the case, its ability to effectively exploit the SDRAM is compromised [11]. For instance, consider the many-core platforms from [1] and [2], which contain processors that rely on caches and that share 4 SDRAM controllers, each managing a 64-bit wide SDRAM module. In such case, a single *read* or *write* command to one of the SDRAM banks transfers 64 bytes of data (a common cache line size) and, hence, there is no need to employ interleaving or to exploit the internal level caching for a single request.

To address the aforementioned scenario, researchers proposed using a combination of bank privatization and *open-row* policy [6], [7], [11]. The former refers to granting a real-time task exclusive access to one or more banks. The latter refers to not flushing the internal level of caching between successive requests. Consequently, the locality of the caching is potentially exploited over the boundary of all requests performed by a task.

Nevertheless, with the new approach, a new challenge was uncovered: the data bus turnaround time. In SDRAMs, a single data bus is shared for *read* and for *write* operations. Hence, SDRAM controllers must enforce a minimum timing interval between the execution of a *read* and of a *write* command (or vice-versa). Such intervals are known as bus turnaround times and are required in order to change the OCT (On-Chip Termination) of SDRAM chips from input to output or from output to input.

As detailed in [12], the faster a SDRAM device is, the larger the corresponding data bus turnaround times are. Hence, the turnarounds pose a challenge that, if not dealt with, lead to poor SDRAM utilisation. In traditional real-time SDRAM controllers, which were discussed in the beginning of this section, such challenge is mitigated because each incoming request is translated into a statically computed bundle of several *read* (or *write*) commands that does not cause a turnaround.

In COTS SDRAM controllers, which are optimized for

• *Leonardo Ecco and Rolf Ernst are with the Technische Universitaet Braunschweig, Germany.*
E-mail: {ecco,ernst}@ida.ing.tu-bs.de

Manuscript received August 1, 2016; revised December 5, 2016.

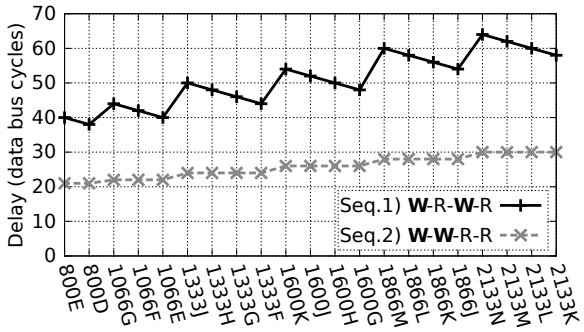


Fig. 1. Minimum distance between the first and the last command of two command sequences in all DDR3 SDRAM devices. The number in each device name represents the speed bin, which is measured in mega transfers per second. The letter represents the device grade.

average performance and rely on the *open-row* policy, the bus turnaround times are mitigated by buffering pending write requests until their number reaches a certain threshold, after which they are served in a batch [13]. Such approach, however, was analyzed in [14] and has been shown as ineffective from the real-time perspective. The reason being that, in order to compute the worst-case latency of a read request, designers must assume a system backlogged with write requests, which leads to pessimistic timing bounds.

In existing *open-row* real-time SDRAM controllers, the turnarounds are either simply accounted for in the timing analysis [6] or are mitigated using multi-rank SDRAM modules [7], [11]. The former demands designers to assume an alternating pattern of interfering *reads* and *writes* in order to compute guarantees, which leads to poor timing bounds. The latter is not cost efficient, as multi-rank modules are expensive. Furthermore, blindly alternating between the ranks is not a scalable solution [15].

Consequently, in [12], we proposed a real-time SDRAM controller that bundles *read* and *write* commands, i.e. minimizes the number of bus turnaround events, thus efficiently tackling the problem in the single-rank domain. To clarify the benefits of read/write bundling, consider Fig. 1, which depicts the minimum distance between the first and the last commands of two different command sequences. Sequence 1 contains an alternating pattern of *writes* and *reads* which requires three bus turnarounds. Sequence 2 contains a bundled pattern, which only requires one bus turnaround. Notice that the bundled pattern is clearly better, being up to 35 cycles faster than the alternating one (for DDR3-2133N).

This article is an **extended version** of [12]. Its **main contributions** in comparison with the original work are:

1) The evaluation performed in [12] was limited to an analytical comparison, i.e. a comparison of latency bounds, with a state-of-the art real-time *open-row* SDRAM controller [6]. In this article, we also consider a real-time *close-row* SDRAM controller [5]. Moreover, we use cycle-accurate simulators to assess how tightly the corresponding timing analyses predict the worst-case behavior of an application.
2) In our original work, our computation of the worst-case latency of *read* and *write* commands relied on a subjective *not-too-late* assumption (better discussed in Sections 3.3.1 and 4.2). With regard to it, we improve the original work on three fronts: firstly, we pinpoint the portion of the analysis that is affected by the assumption. Secondly, we show how a small modification in the analysis computes a worst-case bound without the aforementioned assumption. Finally, we also discuss a small architecture modification to handle *read* and *write* commands that arrive *too late*.
3) We perform a comparison of power consumption trends between the real-time SDRAM controllers under consideration in this article. To our knowledge, this work is the first to compare *open* and *close-row* controllers from the power perspective.

In order to avoid confusion and emphasize the contribution of this article, we also mention that we published a technical report [16] that proposes a multi-generation DDR SDRAM controller that implements read/write bundling. In comparison with this article, the technical report does not consider *close-row* real-time controllers, does not evaluate power consumption and data bus utilisation and omits a discussion about the *not-too-late* assumption and about how data bus turnarounds are addressed by the related work.

The rest of this article is structured as follows: in Section 2, we present the background on SDRAM systems. Then, in Section 3, we describe our SDRAM controller, followed by a timing analysis of it in Section 4. Finally, in Section 5, we present an evaluation of our approach, followed by the conclusion in Section 6.

2 Background on DRAM Systems

In this section, we firstly describe SDRAM memories and their timing constraints and then we discuss the advantages and drawbacks of *open-row* real-time SDRAM controllers. For all intents and purposes, we employ the word SDRAM to refer to DDR2 [17] or DDR3 SDRAM devices [18]. DDR4 SDRAMs [19], which are not yet market dominant, introduced new architectural features. A proper discussion of such features is out of the scope of this article.

2.1 Naming Conventions

Double data rate (DDR) SDRAMs are identified by a string that uses the following pattern: *DDRx-(speed bin)(grade)*. The x stands for the generation, e.g. DDR2 or DDR3. The speed bin is measured in MT/s (mega transfers per second), which corresponds to 2 times the frequency of the data bus measured in MHz (because of the double data rate). For instance, a DDR3-800E device is able to perform at most 800 MT/s and its data bus frequency is equal to 400 MHz.

Finally, the grade, i.e. the letter appended to the end of the string, is used to distinguish between devices that belong to the same speed bin, but that have different timing constraints. The closer to 'A' the grade is, the smaller the timing constraints of the device are and, hence, the faster the device can execute SDRAM commands. For instance, a DDR3-800D can execute SDRAM commands faster than a DDR3-800E, even though both belong to the 800 MT/s speed bin.

2.2 SDRAM Organization, Commands and Constraints

We depict a SDRAM module and the logical structure of a SDRAM chip in Fig. 2. A SDRAM module is a printed circuit board that contains one or more ranks. A rank is comprised of a set of SDRAM chips that share a clock, a command bus and a chip-select signal. Each rank is treated by the SDRAM controller as a single SDRAM chip with a larger number of data bus pins. For instance, in the figure, eight 8-bit data bus chips are used to form a 64-bit wide rank.

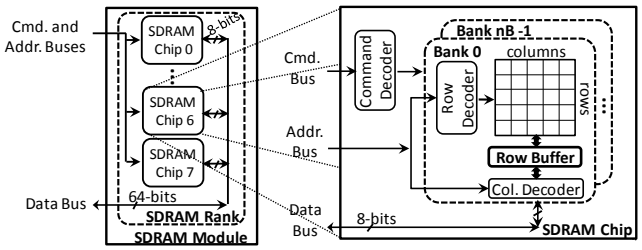


Fig. 2. SDRAM System Organization.

We discuss the logical structure of SDRAM chips. SDRAM chips are divided into banks. We refer to the number of banks in a SDRAM chip as nB. In this paper, we consider DDR2 and DDR3 SDRAMs with nB=8 banks. Each bank contains a matrix-like structure and a row buffer (which is highlighted in the figure). The matrix-like structures are not visible to the memory controller. All data exchanges are instead performed through the corresponding row buffer, which represents the internal level of caching mentioned in the introduction.

There are four commands used to move data into/from a row buffer: *activate*, *precharge*, *read* and *write*. The *activate* (A) command loads a matrix row into the corresponding row buffer, which is known in the literature as *opening a row*. The *precharge* (P) command writes the contents of a row buffer back into the corresponding matrix, which is known in the literature as *closing a row*. The *read* (R) and *write* (W) commands are used to retrieve or forward words from or into a row buffer. We use the acronym CAS (Column Address Strobe) to refer to both *read* and *write* commands.

CAS commands operate in bursts, which means that each of them transfers more than one word. The exact amount of words transferred by a CAS command is determined by the the burst length (BL) parameter. Both DDR2 and DDR3 support BL=8, which is the configuration employed in this article. A single CAS command occupies the data bus for $t_{BURST} = BL/2 = 4$ cycles and transfers $BL \cdot W_{BUS}$ bits, where W_{BUS} represents the width of the data bus.

We discuss timing constraints. There are several timing constraints that dictate how many cycles apart consecutive commands must be. We enumerate them for one DDR2 and one DDR3 devices in Table 1. Notice that there are three different types of constraints: the ones that refer to the minimum distance between commands issued to the same bank (exclusively intra-bank constraints), the ones that refer to the minimum distance between commands issued to different banks (exclusively inter-bank constraints), and the ones that refer to the minimum distance between commands issued to any bank (inter- and intra-bank constraints). The constraints

TABLE 1
Timing constraints for DDR2 and DDR3 devices (available in [17] and [18]), considering a row size of 2KB.

JEDEC DDR2 and DDR3 Specification (data bus cycles)			
Constraint	Description	DDR2-800C	DDR3-1866M
Exclusively intra-bank constraints (same bank)			
t_{RCD}	A to R or W delay	4	13
t_{RP}	P to A delay	4	13
t_{RC}	A to A delay	22	45
t_{RAS}	A to P delay	18	32
t_{WL}	W to data bus transfer delay	3	9
t_{RL}	R to data bus transfer delay	4	13
t_{RTP}	R to P delay	3	7
t_{WR}	End of a W operation to P delay	6	14
Exclusively inter-bank constraints (different banks)			
t_{RRD}	A to A delay	4	6
t_{FAW}	Four *activate* window	18	33
Inter- and intra-bank constraints (any bank)			
$\boldsymbol{t_{WtoR}}$	**W to R delay**	**10**	**20**
$\boldsymbol{t_{RtoW}}$	**R to W delay**	**6**	**10**
t_{WTR}	End of W data transfer to R delay	3	7
t_{RTW}	R to W delay	6	10
t_{BURST}	Data bus transfer	4	4
t_{CCD}	R to R or W to W delay	4	4

that refer to data bus turnarounds fall into the last category. For the interested reader, we provide a graphical depiction of the constraints in Appendix A.

We discuss data bus turnarounds. The DDR2 and DDR3 standards [17], [18] specify two constraints that refer to data bus turnarounds: t_{RTW} and t_{WTR}. The former establishes the minimum distance between a *read* followed by a *write*. The latter establishes the minimum distance between the end of the data transfer of a *write* command and a *read*. To avoid confusion and have constraints with symmetric meanings and notations, we employ the notation t_{RtoW} to refer to the minimum distance between a *read* followed by a *write*, and t_{WtoR} to refer to the minimum distance between a *write* followed by a *read*. Given the textual definition, notice that t_{RtoW} is equal to t_{RTW}, while t_{WtoR} amounts to $t_{WL} + t_{BURST} + t_{WTR}$.

Finally, we discuss refreshes. SDRAMs must be refreshed every $t_{REFI} = 7.8\mu s$ in order to prevent the capacitors that store data from being discharged. This is accomplished with the *refresh (R)* command. The amount of cycles required for a *refresh* to complete (referred to as t_{RFC}) varies according to the SDRAM device, e.g. $t_{RFC} = 36$ cycles for a DDR3-800E.

2.3 Open-Row Real-Time SDRAM Controllers

As discussed in the introduction, this paper concentrates on *open-row* real-time SDRAM controllers. Such controllers only precharge row buffers if a refresh must be executed or if an incoming request needs to access a row currently not present in the corresponding row buffer. Incoming requests are then translated into either a CAS command, in case of a row buffer hit, or into a *precharge-activate*-CAS command sequence, in case of a row buffer miss.

With regard to traditional real-time controllers, which rely on a combination of bank-interleaved address mapping and *close-row* policy, *open-row* SDRAM controllers have two main advantages: firstly, they do not require narrow data

buses and/or large request granularities to be effective. And secondly, they potentially consume less power, as they avoid power-hungry closing and opening of rows.

However, also with regard to traditional real-time controllers, they have two main drawbacks: firstly, they demand a bank privatization setup to be effective, as it prevents different real-time tasks from destroying the row buffer locality of each other. And secondly, due to the bank privatization, they potentially suffer from poor memory utilisation, as a task might not need all the storage provided by a bank.

Finally, it is worth highlighting that in case data exchange between real-time tasks is necessary, one or more banks can be designated for such purpose, i.e. they can be shared. Such strategy has been discussed in [11] and is out of the scope of this article. Moreover, assigning tasks to SDRAM banks can be achieved with a software virtual addressing layer, as discussed in [20].

3 SDRAM Controller Architecture

In this section, we firstly provide an architectural overview of our SDRAM controller and then we discuss in detail the blocks responsible for command scheduling. Before we start our discussion, however, we highlight that our controller supports a single request granularity. Supporting different granularities, which would be necessary for instance if a DMA engine competes for the SDRAM with cache-relying processors, is out of the scope of this article (as it constitutes an orthogonal challenge already investigated in [11]).

3.1 Architectural Overview

We depict the architecture of our SDRAM controller in Fig. 3. Notice that the architecture is comprised of 6 types of blocks: bank address mapping, bank request queues, bank schedulers, command registers, data buffers and channel scheduler. Incoming requests go firstly through the bank address mapping block, which decodes their addresses and forwards them to the proper bank request queue. Requests are then removed one at a time from the queues by the corresponding bank scheduler, whose job is to process them. Processing a request means translating it into a set of SDRAM commands, which are then forwarded to the command registers. Each bank scheduler has its own command register and each command register stores a single command (the oldest outstanding command).

Finally, the channel scheduler, which implements the read/write bundling mechanism, arbitrates between different command registers and sends the selected command to the SDRAM module, i.e. executes it. In the rest of this section, we discuss in detail the SDRAM controller blocks that are responsible for SDRAM command scheduling (depicted in gray in the figure).

3.2 Bank Schedulers and Command Registers

The function of bank schedulers is to translate a memory request into a set of SDRAM commands that fulfill such request. If the bank scheduler employs the *open-row* buffer policy, which is the case considered in this article, a request is translated into either a CAS command or into a *precharge-activate-CAS* sequence, depending on whether

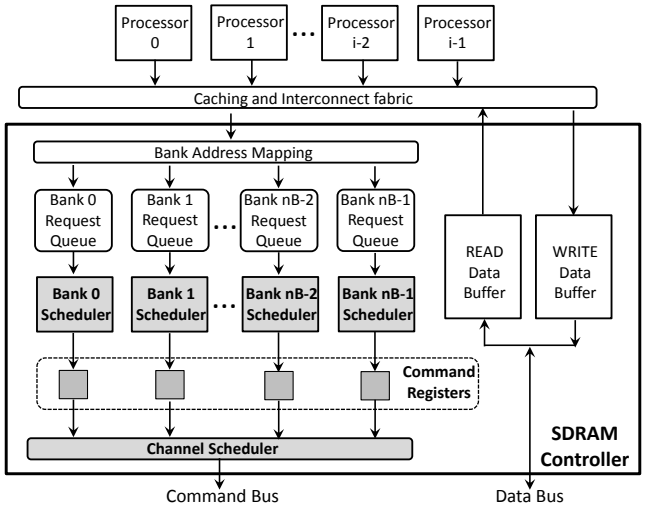


Fig. 3. Logical architecture of our SDRAM controller. The blocks responsible for SDRAM command scheduling are highlighted in gray.

it hits or misses at the row buffer. The function of the command registers is to serve as an intermediate level of buffering that decouples the implementation of the channel scheduler from the bank schedulers. There is one command register for each bank scheduler. The channel scheduler removes commands from the registers when the commands are executed (sent to the SDRAM module). This allows the bank scheduler whose register was emptied to insert a new command (after the pertinent constraints no longer pose a violation), and so on.

A bank scheduler must only place a command in its register if such command can be immediately executed by the channel scheduler without violating any *exclusively intra-bank* timing constraints, i.e. timing constraints that rule the minimum distance between commands issued to the same bank and to the same bank only. For instance, if the channel scheduler executes an *activate* from a command register, then the corresponding bank scheduler must wait at least t_{RCD} cycles before inserting a *write* into the aforementioned command register.

3.3 Channel Scheduler

The channel scheduler has two functions: firstly, to regularly refresh the SDRAM module and, secondly, to arbitrate between and execute commands from the command registers. Because the refresh logic is trivial, in this subsection, we focus on the (non-refresh) command scheduling. As we already discussed, commands placed in the command registers can be immediately executed without violating any *exclusively intra-bank* constraints. Hence, the channel scheduler only needs to prevent the *exclusively inter-bank* and the *intra- and inter-bank* timing constraints.

We depict a block diagram of the channel scheduler in Fig. 4. Notice that commands are arbitrated in two layers. Firstly, they are arbitrated inside their own type arbiters, i.e. CAS commands are routed to the CAS Arbiter and *activate* and *precharge* commands go to the Activate and Precharge Arbiters, respectively. Then, in the second layer of arbitration, i.e. the Command Bus Arbiter, a command that won the arbitration in its type arbiter competes with

interfering commands from other types. We now discuss each of the aforementioned arbiters individually.

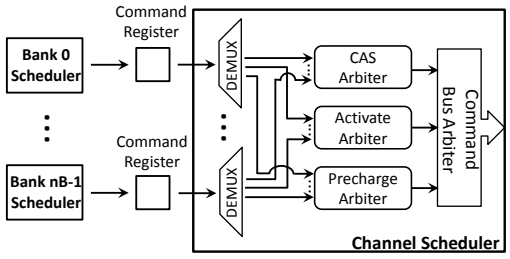


Fig. 4. SDRAM Channel Scheduler. The refresh logic is omitted for the sake of simplicity.

3.3.1 CAS Arbiter

The CAS Arbiter implements the bundling of *read* and *write* commands. For that purpose, it relies on the concept of scheduling rounds, which can last several cycles. Moreover, it schedules commands according to three rules: 1) in each round, at most one CAS command from each of the command registers is executed. 2) In the beginning of each round, the CAS Arbiter selects (if existing) CAS commands that match the type of the last CAS command from the previous round. For instance, in Fig. 5, the CAS Arbiter starts round $i+1$ serving *write* commands, because round i finished with a *write* command. And 3), in each round, at most one *sweep* of *read* commands and one *sweep* of *write* commands is performed. Hence, if a CAS command which is not blocked by the first rule arrives too *late*, e.g. a *read* command arrives after the end of the *sweep* of *read* commands, such command is postponed until the next round.

We make two important observations about the rules. Firstly, they enforce that at most one turnaround happens in each scheduling round. And secondly, the *not-too-late* assumption mentioned in the introduction refers to assuming (in the timing analysis) that any CAS command that is not blocked by the first rule will also not be postponed until the next round due to the third rule.

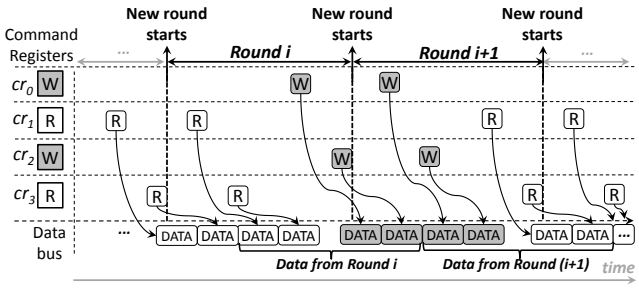


Fig. 5. Example of read/write bundling in a system with nB=4 banks. In the figure, we consider that two command registers provide a continuous stream of *writes* and two other command registers provide a continuous stream of *reads*. Notice that at most one data bus turnaround is required in each round. Notice also that the turnaround causes an idle bubble in the data bus.

To implement bundling, the CAS Arbiter requires a stateful architecture. The state is comprised of a vector of *served flags*, that keep track of which command registers have already been served in the current round, and a *bundling-type* register, that defines which type (*read* or *write*) of CAS commands have currently priority. To clarify how the state is used to perform scheduling, we depict a diagram showing the operation of the CAS Arbiter in Fig. 6. In the figure, it is important to notice that the demultiplexing layer from the channel scheduler (see Fig. 4) enforces that only *read* or *write* commands arrive at the input of the CAS Arbiter. Therefore, the *activate* and *precharge* commands contained in the registers of banks 0, 1 and 7 do not arrive at the input of the CAS Arbiter (their boxes are empty).

We discuss each step performed by the CAS Arbiter. Firstly, the CAS Arbiter masks out the command registers that have already been served in the round. This is performed using the *served flags* vector. Secondly, the arbiter performs CAS masking, which masks out pending CAS commands that do not match the *bundling-type* register. In the figure, the *bundling-type* is *write* consequently, the *read* command from bank 5 is masked out. Thirdly, a round-robin arbiter selects the next CAS command to be executed. Finally, in the last step (called the timing constraints checker), the selected CAS command is simply held until it causes no timing violations. For CAS commands, there are 4 constraints that need to be accounted for: t_{CCD}, t_{BURST}, t_{RtoW} and t_{WtoR}. So, for instance, if the last command executed in the round was a *read* but the next command that the CAS arbiter wants to execute is a *write*, the timing constraints checker holds the pending *write* for t_{RtoW} cycles.

We describe how the CAS Arbiter state is updated. We firstly discuss the *served flags* vector. Every time a CAS is selected and executed by the Command Bus Arbiter, the corresponding bit in the *served flags* vector is set. Furthermore, the vector is cleared every time a new round starts. A new round starts either when all bits of the vector are set, or when all unset bits of the vector belong to command registers that do not have a pending CAS command.

We discuss the *bundling-type* register. The *bundling-type* register defines which type of CAS operation has priority: *reads* or *writes*. Its value is flipped (from *read* to *write*, or from *write* to *read*) if four conditions are simultaneously satisfied. Firstly, there is at least one command register that has a pending CAS command whose type does not match the value of the *bundling-type* register. Secondly, the *served flag* of the command from the first condition is unset. Thirdly, the output of the CAS Masking step is null. And finally, no flipping of the *bundling-type* register has taken place in the current scheduling round. These four conditions enforce that, in each round, at most one data bus turnaround is required, as shown in Fig. 5.

3.3.2 Activate Arbiter

There are two timing constraints that dictate how far apart *activate* commands to different banks must be from each other: t_{RRD} and t_{FAW}. The t_{RRD} dictates the minimum distance between consecutive *activate* commands to different banks. The t_{FAW} is a bit more complex. It establishes a time window in which at most 4 *activate* commands can be executed. As long as t_{RRD} and t_{FAW} are not violated, the Activate Arbiter simply forwards the oldest pending *activate* command to the Command Bus Arbiter.

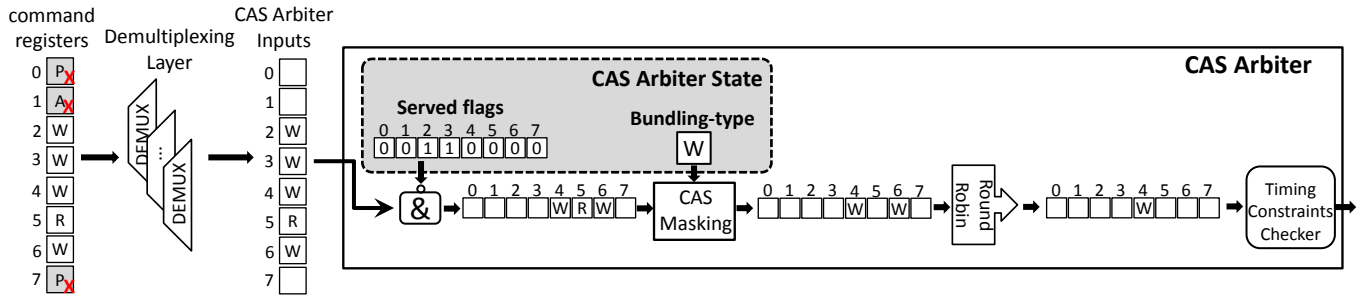


Fig. 6. CAS Arbiter operation for a system with nB=8. For the sake of simplicity, the logic that updates the CAS Arbiter state (*served flags* vector and *bundling-type* register) has been omitted.

3.3.3 Precharge Arbiter

There are no inter-bank timing constraints between *precharge* commands to different banks. For instance, a *precharge* command to bank 0 can be executed one cycle after a *precharge* command to bank 1. Hence, the *precharge* Arbiter simply forwards the oldest pending *precharge* to the Command Bus Arbiter.

3.3.4 Command Bus Arbiter

The command bus can only carry one command per cycle and, hence, needs to be arbitrated. Given that the *exclusively intra-bank* timing constraints are handled by the bank scheduler and that *inter-bank* timing constraints are handled by the type arbiters, this stage of arbitration only needs to select between the output of the type arbiters. For that purpose, it prioritizes the output of the CAS Arbiter. If no pending CAS is available, the oldest non-CAS (*precharge* or *activate*) is given priority. (Here we highlight that in [12], *activates* had priority over *precharges*, which brought no benefit).

4 Timing Analysis

In this section, we describe how to calculate the worst-case cumulative SDRAM latency of a task (L_{Task}^{SDRAM}) using our SDRAM controller, i.e. the maximum amount of time that a task spends idle while waiting for its SDRAM requests to be served. A task, for the sake of this article, is a processor executing a computer program.

We structure our analysis into four parts: firstly, in Section 4.1, we describe our processor, bank scheduler and bank address mapping assumptions. Secondly, in Section 4.2, we compute the worst-case latencies of individual commands. Then, in Section 4.3, we combine the worst-case latencies of individual commands with the delays introduced by the bank scheduler in order to calculate the worst-case latencies of SDRAM requests. Finally, in Section 4.4, we compute L_{Task}^{SDRAM}, i.e. the sum of the worst-case latencies of all SDRAM requests.

4.1 Assumptions

Our timing analysis demands no knowledge about the behavior of interfering tasks on the system. Moreover, it relies on the following assumptions: 1) the processor running the task *under analysis* (u.a.) relies on caches and only accesses the SDRAM to retrieve or forward cache lines. 2) The processor is fully timing compositional [21], which means that it uses in-order execution and stalls at every read request. 3) The *write-buffer* between the cache and SDRAM is disabled and, hence, the processor also stalls at write requests[1]. In the ARMv8-A architecture [22], for instance, this is achieved by disabling the *early write acknowledgment* feature. 4) No multi-threading/context switches occur due to task scheduling. This enforces that no cache related effects change the number of cache misses experienced by the task u.a.. 5) The task u.a. has exclusive access to one of the banks (bank privatization) and the corresponding bank scheduler employs the *open-row* policy. 6) Our lemmas and equations do assume the *not-too-late* behavior mentioned in Section 3.3.1. However, after we present them, we also show how a simple trick can be used to compute a bound that does not rely on such behavior.

4.2 Worst-case Latency of a Command

In this subsection, we calculate the worst-case latency between the insertion of a command into a register and its execution by the channel scheduler. For that purpose, we assume that each command suffers maximum interference within its type arbiter. For instance, when calculating the worst-case latency of an *activate*, we assume all interfering command registers also have pending *activate* commands.

We make the following observations about our discussion: firstly, to avoid confusion, we refer to the command register that holds the command u.a. as cr. Secondly, we refer to each of the nB-1 interfering command registers as icr_i, where i is an index. Thirdly, to save space, the figures in this subsection are depicted with nB=4 banks, even though DDR3 devices have nB=8 banks. Fourthly, because different commands are subject to different timing constraints and have different priorities inside the channel scheduler, we calculate the worst-case latency individually for *read*, *activate* and *precharge*. The case for a *write* command is similar in a symmetrical fashion to the one for a *read* command and, hence, only discussed in Appendix B.

Finally, for CAS commands, i.e. *read* or *write* commands, we further distinguish between two types of worst-case latency: 1) the one experienced if the CAS u.a. succeeds, i.e. follows, a CAS command in cr (SC). And 2) the one experienced if the CAS u.a. succeeds a non-CAS command

1. Notice that the presence of a *write-buffer* allows a processor to keep executing while a write request is being processed, potentially hiding the latency of a write request. Consequently, making a no *write-buffer* assumption is conservative.

in cr (SNC). As it will become clear, this distinction allows us to properly analyze the round-oriented operation of the CAS Arbiter. We summarize the notations used for worst-case latencies of commands in Table 2.

TABLE 2
Notation used for worst-case latencies of SDRAM commands.

.

Worst-case Latency Notation	SDRAM Command u.a.	Succeeding a	Computed according to
L^R_{SC}	R	R or W	Theorem 1
L^R_{SNC}	R	A	
L^W_{SC}	W	R or W	Theorem 5 (Appendix B)
L^W_{SNC}	W	A	
L^A	A	P	Theorem 2
L^P	P	P or R or W	Theorem 3

We now calculate the worst-case latency of a *read*. For that purpose, we firstly describe an expression that computes the latency of a *read* as a function of t_{DELAY}, i.e. the distance between the insertion of the *read* u.a. into cr and the execution of the previous CAS command that occupied cr. We refer to such expression as $L^R(t_{DELAY})$. Both L^R_{SC} and L^R_{SNC} are then computed using it with different parameters.

We depict a scenario that induces the worst-case latency of a *read* in Fig. 7. The intuition is that the *read* u.a. is potentially blocked twice by each interfering command register (depending on how small t_{DELAY} is). Moreover, we assume that data bus turnarounds happen as often as possible, i.e. one at every scheduling round. With that in mind, we state Lemma 1.

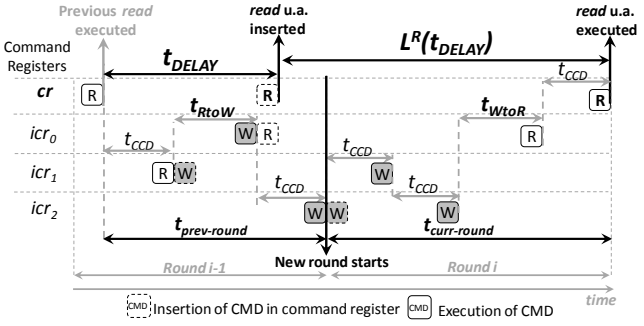


Fig. 7. Worst-case latency of a *read* command that succeeds a CAS command in cr for a system with nB=4 banks.

Lemma 1. *The worst-case latency of a* read *(that is inserted into cr t_{DELAY} cycles after the previous CAS that occupied cr is executed) is calculated with Eq. 1.*

$$L^R(t_{DELAY}) = \max\{t_{prev\text{-}round} - t_{DELAY}, 0\} + t_{curr\text{-}round} \quad (1)$$

where:

$$t_{prev\text{-}round} = (nB - 2) \cdot t_{CCD} + t_{RtoW} \quad (2)$$

$$t_{curr\text{-}round} = (nB - 1) \cdot t_{CCD} + t_{WtoR} \quad (3)$$

Proof. It is trivial to observe that, in order to maximize $L^R(t_{DELAY})$, the previous CAS that occupied cr should be the first to be served in its scheduling round and that the *read* u.a. should be the last to be served in its scheduling round. Moreover, to enforce one turnaround in each of the scheduling rounds, the previous CAS in cr must also be a *read*. Consequently, the rest of this proof consists in addressing the correctness of Eq. 1.

The worst-case latency of a *read* command is given by the sum of the blocking experienced by it in two distinct and consecutive rounds of the CAS Arbiter (see Figure 7): firstly, in *round* $i-1$, i.e. the round in which the previous *read* that occupied cr is executed, and then in *round* i, i.e. the round in which the *read* u.a. is executed.

We firstly discuss the blocking in *round* $i-1$. In *round* $i-1$, the amount of blocking experienced by the *read* u.a. amounts to $\max\{t_{prev\text{-}round} - t_{DELAY}, 0\}$ cycles. The $t_{prev\text{-}round}$ portion is computed according to Eq. 2 and represents an upper bound on the time required for *round* $i-1$ to finish because: 1) all *icrs* provide an interfering CAS, and 2) one data bus turnaround is required. The $t_{prev\text{-}round}$ is then subtracted by the t_{DELAY}, which can be inferred from our assumption of a processor stalls at every request.

We discuss the blocking in *round* i. In *round* i, i.e. the round in which the *read* u.a. is executed, the blocking suffered by the *read* u.a. amounts to $t_{curr\text{-}round}$, which is calculated according to Eq. 3. Again, the equation computes an upper bound on the blocking experienced by the *read* u.a. in *round* i because: 1) all *icrs* provide an interfering CAS command. And 2) a turnaround is required. This concludes, by construction, the proof of correctness of Eq. 1. □

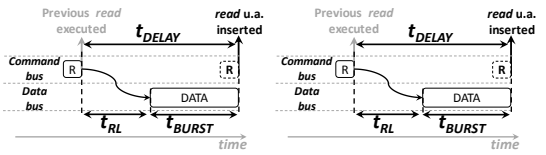
From the definition of $L^R(t_{DELAY})$, we can derive Theorem 1.

Theorem 1. *The worst-case latency of a* read *is given by Eq. 4 if it succeeds a CAS, and by Eq. 5 if it succeeds a non-CAS.*

$$L^R_{SC} = L^R(t_{RL} + t_{BURST}) \quad (4)$$

$$L^R_{SNC} = L^R(t_{RL} + t_{BURST} + t_{RP} + t_{RCD}) \quad (5)$$

Proof. Both Eqs. 4 and 5 use the function from Lemma 1 with different values of t_{DELAY}. For the calculation of L^R_{SC}, i.e. the worst-case latency of a *read* that succeeds a CAS in cr, we know t_{DELAY} is at least $t_{RL} + t_{BURST}$ cycles (as depicted in Fig. 8a), because of our assumption of a processor that tolerates at most one outstanding request. However, for the calculation of L^R_{SNC}, we know there is an *activate* and a *precharge* before consecutive CAS commands. Hence, we add the corresponding latencies, as depicted in Fig. 8b. □



(a) For computing L^R_{SC}. (b) For computing L^R_{SNC}.

Fig. 8. Graphical depiction of t_{DELAY}. (Proof aid for Theorem 1).

We now discuss the *not-too-late* assumption. For that purpose, consider the example from Fig. 7 and the equation from Lemma 1. Notice that if t_{DELAY} is sufficiently large, the equation will only account for the blocking experienced

in a single scheduling round (given by $t_{curr\text{-}round}$). However, this only remains accurate if the third operational rule from the CAS Arbiter (see Section 3.3.1) is never invoked.

In theory, it is possible to carefully handcraft a scenario in which the third rule must be used by the CAS Arbiter. This is demonstrated in Fig. 9. In practice, cycle-accurate simulations have proven such scenario to be quite unlikely. Hence, it can be safely ignored, given that the end-result of our analysis is computed over all requests from a task, from which the vast majority will not experience the *too late* behavior of a *read* command. (Alternatively, one could employ FCFS instead of round-robin in the CAS Arbiter and modify its second rule of operation so that if a CAS arrives too late, then its type determines how the new scheduling round starts, e.g., in Fig. 9, round i would start executing the read u.a.. This would have minimum impact in Lemma 1).

Assuming no modifications in the CAS Arbiter, a bound independent of the *not-too-late* assumption can be calculated by enforcing that the $L^R(t_{DELAY})$ function is only used with $t_{DELAY} = 1$, i.e. enforcing that $L^R_{SNC} = L^R_{SC} = L^R(1)$. The number one is used instead of zero because, in order to construct the scenario, we must assume that the *read* u.a. is inserted into cr one cycle after the decision to end the sweep of *read* commands is made.

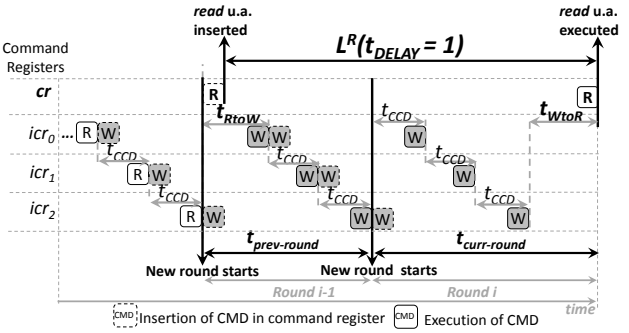


Fig. 9. An example of a *read* arriving *too late* for scheduling.

We now discuss the worst-case latency of *activates* and *precharges*. For that purpose, we firstly state Lemma 2, which captures the effect of the lower priority of *activates* and *precharges* with regard to CAS commands.

Lemma 2. *Given a sequence of* n precharge *and/or* activate *commands that can be immediately executed without violating timing constraints and that can only postpone each other for one cycle due to command bus contention, the maximum timing interval (in cycles) required to execute such sequence is given by Eq. 6.*

$$\alpha_{PA}(n) = n + \left\lceil \frac{n}{t_{BURST} - 1} \right\rceil \tag{6}$$

Proof. Activate and *precharge* commands have lower priority than CAS commands. However, any two consecutive CAS commands must be executed at least t_{BURST} cycles apart (or by even more cycles if a data bus turnaround is required). Hence, in any interval of t_{BURST} cycles, at least $t_{BURST} - 1$ cycles will be free for the execution of *activates* and *precharges*. Eq. 6 comes directly from such observation (for the interested reader, a graphical depiction is available in Appendix D). □

We now compute the worst-case latency of an *activate* command.

Theorem 2. *The worst-case latency of a* activate *command is calculated using Eq. 7.*

$$L^A = (t_{FAW} - 4 \cdot t_{RRD}) \quad + \max\{exp1, exp2\} \tag{7}$$

where:

$$exp1 = (nB - 1) \cdot t_{RRD} + (nB - 1) \cdot \Delta_A \tag{8}$$

$$exp2 = exp1 + (t_{FAW} - (4 \cdot t_{RRD} + 3 \cdot \Delta_A)) \cdot K \tag{9}$$

$$\Delta_A = \alpha_{PA}(1) - 1 \tag{10}$$

$$K = \left\lfloor \frac{(nB - 1)}{4} \right\rfloor \tag{11}$$

Proof. In order to assist our proof, we depict an example of the worst-case latency of an *activate* in Fig. 10. Notice that the figure has three main features: (1) it considers that four *activates* are executed *as-late-as-possible* before the insertion of the *activate* u.a.. (2) When the *activate* u.a. is inserted into cr, each of the $nB - 1$ interfering banks has an older pending *activate*. (3) After *activate* command(s) from interfering bank(s) are executed, such bank(s) provide higher-priority CAS commands.

We now discuss why such features lead to the worst-case. The first feature forces us to account for a residual latency (consequence of t_{FAW}). The second feature comes from the observation that a CAS or *precharge* command in an interfering bank can only postpone the *activate* u.a. by one cycle (due to data bus contention), while an *activate* in an interfering bank can postpone the *activate* u.a. by at least t_{RRD} cycles. The last feature enforces that *activates* are blocked as often as possible by higher-priority CAS commands (by as often, we mean as long as the second feature is not compromised).

That being said, we now prove the correctness of the equation that computes L^A, which has two main terms. The leftmost term accounts for the residual latency mentioned in the first condition. The rightmost term (max operator) accounts for the remaining latencies by selecting the largest value between two expressions. The first one ($exp1$) considers that t_{FAW} is hidden by the occurrences of t_{RRD} and the blocking due to higher-priority CAS commands (which is not the case in Fig. 10). The second one ($exp2$) considers that t_{FAW} is not hidden and basically just replaces $(4 \cdot t_{RRD} + 3 \cdot \Delta_A)$ by t_{FAW} in $exp1$ for each of the K times in which the t_{FAW} constraint is activated. □

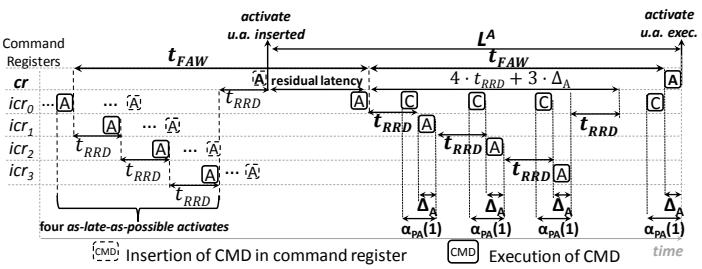


Fig. 10. Worst-case latency of an *activate* command in a hypothetical system in which nB=5. The letter C refers to a CAS command.

Finally, we compute the worst-case latency of *precharge* commands with Theorem 3.

Theorem 3. *The worst-case latency of a* precharge *command is calculated using Eq. 12.*

$$L^P = \alpha_{PA}(nB) \tag{12}$$

Proof. Precharge commands can be executed back-to-back (one per cycle). In the worst-case, the *precharge* u.a. is blocked once by older non-CAS commands in interfering banks. Moreover, interfering non-CAS commands can be blocked higher-priority CAS commands once every t_{BURST} cycles. Consequently, we compute L^P by invoking α_{PA} with nB as argument. Notice that we employ nB (instead of the $nB-1$ interfering non-CAS commands) as an argument to the $\alpha_{PA}(n)$ function, as we also account for one cycle required to execute the *precharge* u.a.. □

4.3 Worst-case Latency of a Request

We define the worst-case latency of a request as the time between the request arriving at the SDRAM controller and the corresponding data transfer being completed. This latency is influenced by three factors: 1) The latencies imposed by the bank scheduler, which enforce that a command is only inserted into the corresponding command register if it can be immediately executed without violating any *intra-bank* timing constraints. 2) The commands required to fulfill the request and their corresponding worst-case latencies, which were calculated in the previous subsection. 3) The cycles required to perform the data transfer.

Because of the *open-row* policy, the commands required to fulfill a request depend on whether it hits or misses at the row buffer and whether it is a read or a write. Hence, we classify requests into four different types, as described in Table 3. For instance, a read request that does not target a currently opened row is referred to as a *Read Miss (RM)*, it requires a P-A-R command sequence to be fulfilled and its worst-case latency is given by L^{RM}_{Req}. In this subsection, we compute the worst-case latency for read misses and read hits (the case for write misses and write hits is similar and, hence, is only provided in Appendix C).

TABLE 3
Request classification.

Type	Command Sequence	Mnemonic	Latency
Read Miss	P-A-R	RM	L^{RM}_{Req}
Read Hit	R	RH	L^{RH}_{Req}
Write Miss	P-A-W	WM	L^{WM}_{Req}
Write Hit	W	WH	L^{WH}_{Req}

We discuss L^{RM}_{Req}. For ease of comprehension, we depict the factors that contribute to L^{RM}_{Req} in Figure 11a. Notice that, in the figure, there is a number below every latency that contributes to L^{RM}_{Req}. These numbers correlate each latency with one of the factors described in the beginning of this section. For instance, L^P, L^A and L^R_{SNC} are command latencies (factor 2) and, hence, have a 2 below them. Also, observe that, in the worst case, the request u.a. arrives exactly after the previous request was served. This is a direct result of our timing compositional processor assumption. Finally, notice that we use a pattern of white and gray to depict the previous request. Such color scheme is employed to represent that the previous request can be either a read or a write request (and, hence, the letter C, which stands for CAS, is used inside the command box).

We firstly describe $t_{Residual}$. To fulfill a RM request, the bank scheduler firstly needs to precharge the row buffer. However, in order to enforce that no t_{WR} or t_{RAS} violations occur (see Table 1), the bank scheduler needs to delay the insertion of the required *precharge* into the command register by $t_{Residual}$ cycles. To calculate $t_{Residual}$, we consider both the case in which the previous request was a read and the case in which the previous request was a write, as displayed in Eqs. 13, 14 and 15.

$$t_{Residual} = \max\{t^{prev\text{-}R}_{Residual}, t^{prev\text{-}W}_{Residual}\} \tag{13}$$

where:

$$t^{prev\text{-}R}_{Residual} = \max\{t_{RAS} - (t_{RCD} + t_{RL} + t_{BURST}), 0\} \tag{14}$$

$$t^{prev\text{-}W}_{Residual} = t_{WR} \tag{15}$$

If the previous request was a read, then $t_{Residual}$ is a consequence of the t_{RAS} constraint and is given by Eq. 14, which comes directly from the semantics of the used constraints. If the previous request was a write, then $t_{Residual}$ is given by Eq. 15, which simply corresponds to the write recovery time.

We highlight that in order to compute the latency of the request u.a. independently from the previous request, we conservatively employ the $\max$ function in Eq. 13 to select between the largest of the two cases. In the next subsection, when we combine the latencies of all requests to extract guarantees for a task, we describe a correction term that compensates for this overly conservative assumption).

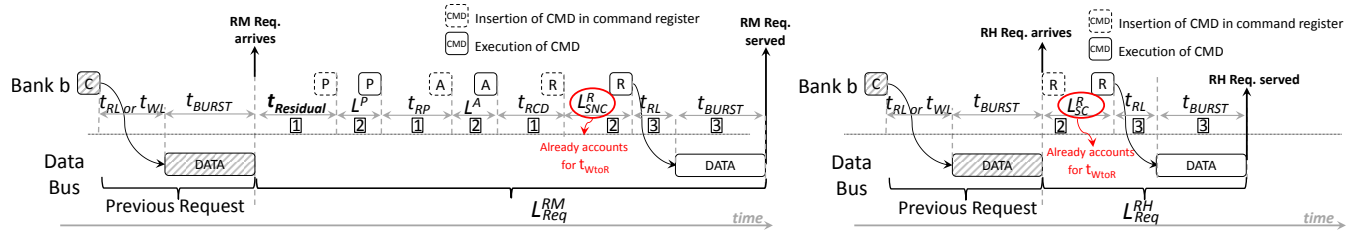
We now discuss the computation of L^{RM}_{Req}. As we already discussed, there are three factors contributing to L^{RM}_{Req}. In order to compute L^{RM}_{Req}, we simply add all three factors. This is formalized with Lemma 3.

Lemma 3. *The worst-case latency of a RM request is given by Eq. 16. In the equation, the leftmost, the middle and the rightmost portions compute the influence of factors 1, 2 and 3, respectively.*

$$L^{RM}_{Req} = (t_{Residual} + t_{RP} + t_{RCD}) + (L^P + L^A + L^R_{SNC}) + (t_{RL} + t_{BURST}) \tag{16}$$

Proof. Eq. 16 is trivial, as it simply sums the command latencies, the bank scheduler delays and the time required to perform a data transfer. □

We compute L^{RH}_{Req}. A RH request only requires a *read* command and, hence, its worst-case latency is simpler, as depicted in Figure 11b. As soon as the request arrives, the bank scheduler inserts the *read* u.a. into the command register. The *read* u.a. is executed after at most L^R_{SC} cycles and the data transfer is completed after $t_{RL}+t_{BURST}$ cycles. Notice that, because no *precharge* is required, the $t_{Residual}$ latency does not need to be taken into account. Moreover, a possible data bus turnaround is already accounted for inside L^R_{SC}. These observations trivially yield Lemma 4.



(a) L^{RM}_{Req} (b) L^{RH}_{Req}

Fig. 11. Decomposition of L^{RM}_{Req} and L^{RH}_{Req}. Notice that for the calculation of L^{RM}_{Req} we employ L^{R}_{SNC}, while for L^{RH}_{Req} we employ L^{R}_{SC}.

TABLE 4
Information required to calculate L^{SDRAM}_{Task}. The second portion of the table is derived from the first portion and is only employed to calculate the correction term for the overly conservative computation of $t_{Residual}$ (see Eq. 13 in Section 4.3).

Notation	Description	
N^{RM}_{Task}	Number of Read Misses from the task	
N^{RH}_{Task}	Number of Read Hits from the task	
N^{WM}_{Task}	Number of Write Misses from the task	
N^{WH}_{Task}	Number of Write Hits from the task	
Notation	**Description**	**Value**
N^{Misses}_{Task}	Number of Misses from the task	$N^{RM}_{Task} + N^{WM}_{Task}$
N^{Reads}_{Task}	Number of Reads from the task	$N^{RM}_{Task} + N^{RH}_{Task}$
N^{Writes}_{Task}	Number of Writes from the task	$N^{WM}_{Task} + N^{WH}_{Task}$

Lemma 4. *The worst-case latency of a RH request is given by Eq. 17. In the Equation, the leftmost and the rightmost portions compute the influence of factors 2 and 3, respectively.*

$$L^{RH}_{Req} = (L^{R}_{SC}) + (t_{RL} + t_{BURST}) \tag{17}$$

4.4 Worst-case Cumulative SDRAM Latency of a Task

We define the worst-case cumulative SDRAM latency of a task (L^{SDRAM}_{Task}) as the maximum amount of time that a task spends idle waiting for its SDRAM requests to be served. For that purpose, we assume that we know the pattern of SDRAM requests performed by a task that leads to its worst-case latency. By pattern, we mean the information enumerated in Table 4. Because computing such pattern is out of the scope of this article (we refer the interested reader to [23]), we extract the pattern from execution traces. We highlight that the same assumption has been made in [11], [12], which also employed a trace-based approach.

In order to compute L^{SDRAM}_{Task}, we firstly multiply the numbers from the first portion of Table 4 by the corresponding request latencies. Then, we correct the overly conservative result (which is a consequence of Eq. 13 in Section 4.3) with a correction term. We formalize the computation of L^{SDRAM}_{Task} in Theorem 4.

Theorem 4. *The worst-case cumulative SDRAM latency of a task is given by Eq. 18.*

$$L^{SDRAM}_{Task} = L^{Reqs}_{Task} - t^{Correction}_{Residual} \tag{18}$$

where:

$$L^{Reqs}_{Task} = (N^{RM}_{Task} \cdot L^{RM}_{Req}) + (N^{RH}_{Task} \cdot L^{RH}_{Req}) + (N^{WM}_{Task} \cdot L^{WM}_{Req}) + (N^{WH}_{Task} \cdot L^{WH}_{Req}) \tag{19}$$

$$t^{Correction}_{Residual} = \begin{cases} 0 \ \ if\ N^{Misses}_{Task} \leq N^{Writes}_{Task}; \\ aux \ \ otherwise. \end{cases} \tag{20}$$

$$aux = (N^{Misses}_{Task} - N^{Writes}_{Task}) \cdot (t^{prev\text{-}W}_{Residual} - t^{prev\text{-}R}_{Residual}) \tag{21}$$

Proof. The first term of Eq. 18, i.e. L^{Reqs}_{Task}, is a direct consequence of our assumption of a processor that stalls at every request. Hence, what remains to be proven is whether the second term, i.e. the correction term $t^{Correction}_{Residual}$, is valid.

For that purpose, we firstly highlight that $t_{Residual}$ is always given by $t^{prev\text{-}W}_{Residual}$, i.e. $t^{prev\text{-}R}_{Residual}$ is always smaller. Moreover, we also highlight that to calculate L^{Reqs}_{Task}, the term $t_{Residual}$ is summed N^{Misses}_{Task} times, i.e. one time for every RM or WM request. In other words, when computing L^{Reqs}_{Task}, we assume that every RM or WM request is preceded by a write (WH or WM) request.

If there are more write requests than requests that miss in the row buffer, such assumption is valid and, hence, $t^{Correction}_{Residual} = 0$. However, if there are less write requests than requests that miss in the row buffer, we know that when computing L^{Reqs}_{Task}, we incorrectly assumed $t_{Residual} = t^{prev\text{-}W}_{Residual}$ for exactly $(N^{Misses}_{Task} - N^{Writes}_{Task})$ requests. In such case, we correct the overly conservative computation with $t^{Correction}_{Residual} = aux$, where aux is given by Eq. 21. □

Notice that, for the sake of simplicity, our theorem purposely disregards the effect that refreshes have in L^{DRAM}_{Task}. This is because, as discussed in [24], the effect of refreshes is negligible in comparison with other command delays, provided that the execution time of the task u.a is not too short. For tasks that fall into the short scenario, a software approach for predictable refreshes is available at [25].

5 Evaluation

In this section, we compare our approach with two different real-time SDRAM controllers:

- the controller from Wu et al. [6].
- the *Analyzable Memory Controller (AMC)* [5].

The SDRAM controller from Wu et al. also uses the *open-row* policy and its analysis assumes that the task under

consideration has exclusive access to one of the banks, i.e. it considers bank privatization. Its main difference in comparison with ours is that older CAS commands always have priority over newer ones, regardless of whether they force a bus turnaround or not.

The AMC employs *close-row* policy and originally relied on a *interleaved* address mapping. However, with a 64-bit data bus (the scenario considered in this evaluation), an *interleaved* address mapping is not useful when SDRAM requests have the size of a cache line. Hence, in order to perform a comparison with our approach, we adopt the strategy employed in [6]: we implement AMC with a bank privatization setup in which each incoming request ultimately is translated into a static command group containing an *activate*-(CAS with Auto-Precharge) sequence. The oldest pending command group, regardless of whether it forces a bus turnaround or not, is given priority.

We discuss the evaluation. Our evaluation is based on SDRAM request traces obtained using the Gem5 platform [26]. The traces are then employed to calculate analytical timing bounds, i.e. worst-case cumulative latencies (L_{Task}^{SDRAM}), which rely on the first five assumptions from Section 4.1[2]. Moreover, the traces are also used as stimuli for cycle accurate simulators of our controller and the other two controllers under consideration in this section. From the simulators, we obtain the following information: 1) the observed cumulative worst-case latency of each application, which we compare with the corresponding analytical bound, 2) the data bus utilisation that each controller is able to maintain, which allows us to compare scheduling efficiency, and 3) SDRAM command traces, which serve as input for a power estimation tool (DRAMPower [27]).

5.1 Application Traces

In our evaluation, we employ a total of eight applications from Mibench [28], a number that matches the quantity of SDRAM banks present in the modules under consideration. In order to collect the traces, the applications are executed in Gem5 in isolation with a 1 GHz timing compositional ARM processor (see Section 4.1) that relies on a 64-kb L1-cache with a line size of 64 bytes. The ratio of request types (see Table 4) exhibited by each application is depicted in Fig. 12. Notice that the profile of the applications varies greatly in terms of row buffer hit ratio and number of write requests.

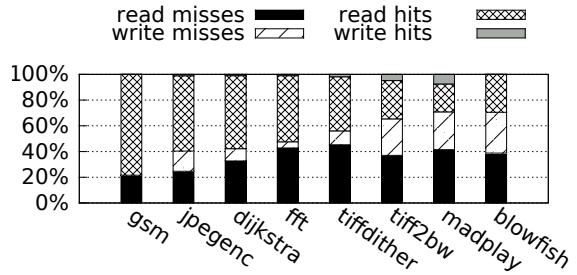


Fig. 12. Percentage of request types by each application.

2. The sixth assumption (*not-too-late* behavior) is only relevant for our controller. We compute analytical bounds with and without it.

We make one important remark about the collected traces. Because the number of requests of each application varies drastically, we *summarize* the traces. More specifically, we generate artificial traces, each containing 5000 requests, but respecting the proportions depicted in Fig. 12. Moreover, when *executing* the traces in simulation, we eliminate time intervals between successive requests, i.e. for each application, we inject a new request as soon as the previous one is served[3]. We perform these steps because equalizing the number of requests and eliminating inter-request time maximizes the interference that each application can exert on each other during simulation.

5.2 SDRAM Modules

For our evaluation, we consider four SDRAM modules manufactured by Micron [29], which are enumerated in Table 5. All modules have 64-bit wide data buses and and were selected according to their commercial availability at the time of writing. The data sheets for the selected modules, which contain the electrical parameters used by the DRAMPower tool, can be retrieved using the corresponding *Part Numbers*.

TABLE 5
Specification of SDRAM Modules from Micron [29].

DDR Gen.	Model	Capacity	Part Number	Die Rev.	Vdd
DDR2	800C	1 GB	MT8HTF12864A(I)Z-80E	G	1.80 V
DDR3	1333H	1 GB	MT8KTF25664AZ-1G4	K	1.35 V
DDR3	1600K	1 GB	MT8KTF12864AZ-1G6	J	1.35 V
DDR3	1866M	2 GB	MT4KTF25664AZ-1G9	P	1.35 V

5.3 Comparison of Worst-case Performance

We now compare the analytical bounds and the latencies observed in simulation for every combination of SDRAM module and SDRAM controller investigated in this article. For our controller, we compute two different analytical bounds: one that relies on the *not-too-late* assumption and one that does not (see Sections 3.3.1 and 4.2). For the other controllers, we employ the timing analyses presented in the corresponding papers. As for the observed latencies, we perform the following procedure: for each combination of SDRAM controller and SDRAM module, we *execute* all eight application traces simultaneously in the corresponding controller simulator and measure the delays that each trace experiences.

We present the results in Figs. 13a, 13b, 13c and 13d. In the figures, solid colors are used to represent analytical bounds, while pattern fills represent latencies observed in simulation. Moreover, for each application, results are normalized to the analytical bound for AMC. From the analytical perspective, we observe the following trends:

1) For the *open-row* controllers, applications that have a larger number of row buffer hits have smaller

3. Notice that such setup (of injecting a request as soon as the SDRAM controller acknowledges the service of the previous request) limits the gains that could be achieved with a *write-buffer*. As a matter of fact, modifying the SDRAM controller simulators in order to perform *early acknowledgment* of write requests (see Section 4.1) brought negligible changes in the simulation results.

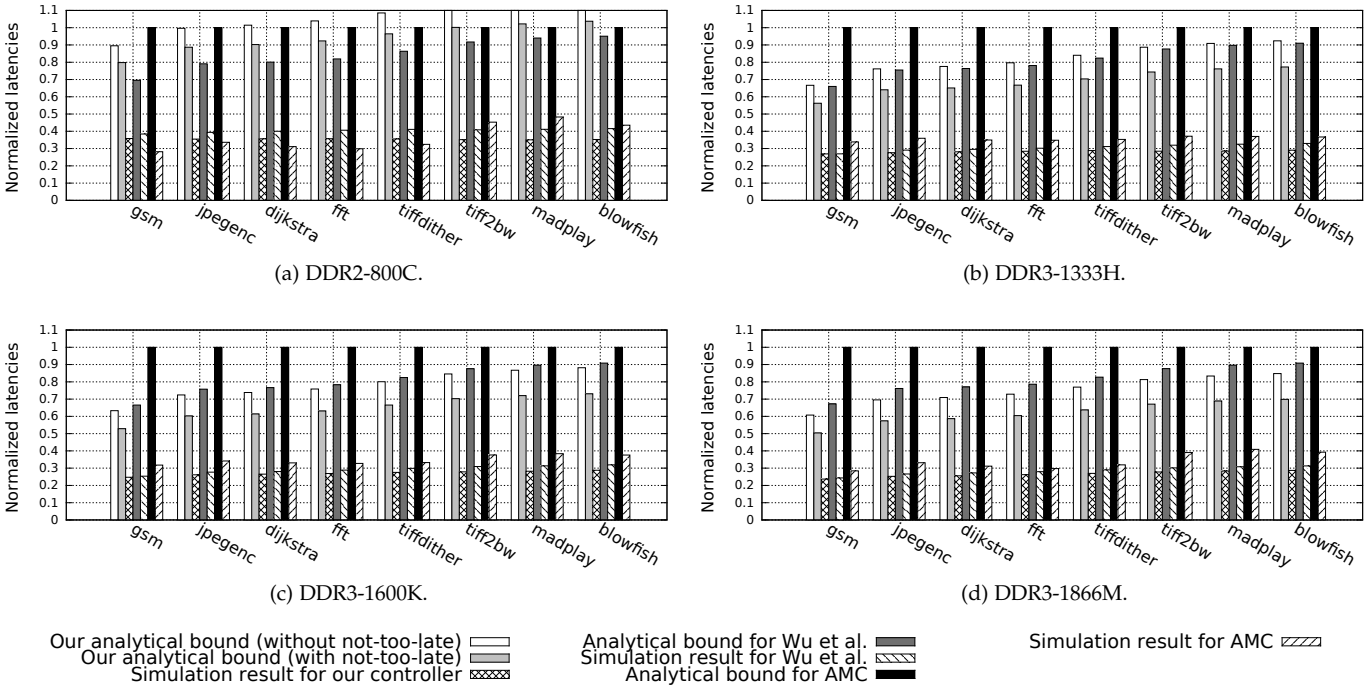


Fig. 13. Comparison of worst-case cumulative latencies. For every application, the results are **normalized** to the analytical bound obtained for AMC. Hence, the smaller the bar, the better the result. For our controller, we compute bounds both assuming and not assuming the *not-too-late* behavior.

timing bounds than the ones with a low number of row buffer hits. Moreover, the advantage of *open-row* controllers over *close-row* ones is larger in high-speed modules, e.g. compare the results obtained for DDR2-800C with the ones for DDR3-1866M. (For the DDR2-800C, our controller provides worse bounds than AMC for most of the applications). This is because in high-speed modules, the overhead for closing and opening rows is larger (see Table 1).

2) The advantage of our controller is better highlighted in high-speed modules. This is because, in the worst-case scenario, a CAS command in our controller can be potentially blocked twice by other CAS commands in each of the interfering banks (see Fig. 9), while in the other controllers, a CAS command can only be blocked once by CAS commands in interfering banks. Consequently, in order for the CAS command reordering to pay-off, the overhead for data bus turnarounds must be large, which is the case in high-speed modules such as the DDR3-1866M (moreover, we also provide results for a DDR3-2133N module in Appendix E).
3) As expected, the bounds provided by our controller are better is we assume the subjective *not-too-late* behavior. This is because if such assumption is made, a portion of the interference suffered by a CAS command is hidden by t_{DELAY}.

From the perspective of latencies observed in simulation, we observe the following trends:

1) The observed latencies for AMC vary according to the number of write requests in the application under analysis. The larger the number of write requests, the larger is the probability that the application forces a bus turnaround (hence, experiencing a larger delay). Such effect would be hidden in systems with narrow data buses because of the static bundling of *reads* and *writes* (see Section 2.3).
2) For some applications and the DDR2-800C module, the AMC actually provides better observed latencies than *open-row* controllers. This is because for slower SDRAM devices, the overhead to close and open rows is smaller (see Table 1). For faster modules, e.g. DDR3-1866M, such overhead increases and, hence, the *open-row* controllers have an advantage.
3) When comparing the observed latencies of *open-row* controllers, our controller displays a small advantage over the one from Wu et al.. The advantage is not larger because, in order for read/write bundling to improve performance over Wu et al., there must be two or more pending *write* commands in one scheduling round. Because the number of write requests is limited (see Fig. 12), such scenario does not correspond to the common case.

5.4 Data Bus Utilisation

From the simulations performed in the last subsection, we extract SDRAM command traces. Each command trace contains commands from requests belonging to all eight applications. Scrutinizing the quantity and time stamp of all commands in a trace, we can compute the average data bus utilisation that each controller can maintain. We depict the results in Figs. 14a, 14b, 14c and Fig. 14d.

Notice that the figures measure time in data bus clock cycles (and not in nanoseconds). Hence, even though using

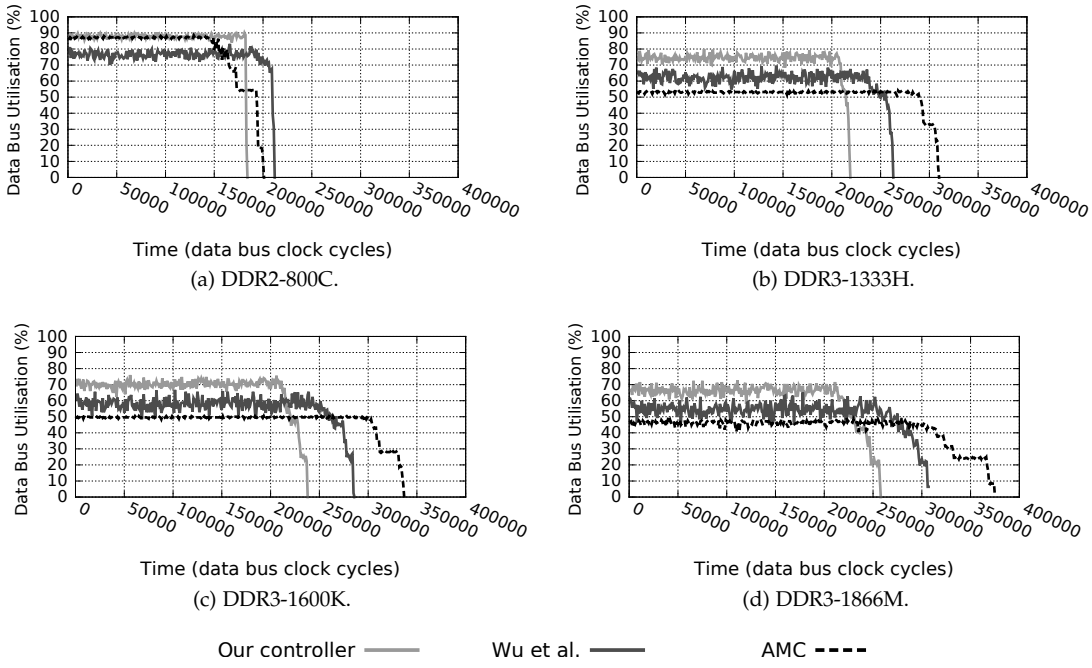


Fig. 14. Comparison of data bus utilisations. The higher the utilisation, the more efficient the SDRAM controller. The drop to 0% of utilisation marks the moment in which all requests from the workload were served.

a DDR2-800C takes less data bus cycles than a DDR3-1866M to serve the same set of requests, the latter takes less nanoseconds, as it has a clock period of only 1.07 ns, while the former has a period of 2.5 ns.

We observe the following trends:

1) For SDRAMs with higher operating frequencies, it becomes harder to keep high data bus utilisations, e.g. compare Fig. 14a and Fig. 14d. This is because the timing constraints are larger for them.
2) For DDR2-800C, AMC actually performs better than the controller from Wu et al.. As a matter of fact, its utilisation mostly overlaps with the one displayed by our controller. Again, this is because the overhead to close and open rows is smaller for devices with reduced operating frequency.
3) For the remaining modules, AMC performs worse than the controller from Wu et al.. Moreover, regardless of the SDRAM module, our controller consistently maintains higher utilization than the other two investigated controllers.

5.5 Power Consumption

Finally, using the same command traces employed in the last subsection, we compute a power consumption estimate using the DRAMPower tool [27]. The results are depicted in Fig. 15 and represent the total amount of energy (in micro joules) required to serve all requests.

We observe the following trends:

1) Regardless of the controller, DDR2-800C has by far the worst power consumption. This is because DDR2 memories are simply not as energy efficient as DDR3. Moreover, they have an operating voltage of 1.8 V (see Table 5), against 1.35 V of the DDR3 investigated modules.
2) For all SDRAM modules, the AMC consumes more power than *open-row* controllers. This is because closing and opening rows is an energy-costly operation.
3) For all SDRAM modules, our controller provides a small power consumption reduction over the controller from Wu et al.. This is because it serves the requests of a workload faster (see Figs. 14a, 14b, 14c and Fig. 14d). Hence, the amount of static power dissipated is smaller.

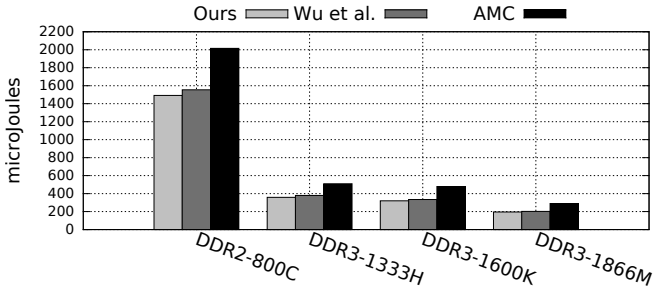


Fig. 15. Power consumption.

6 Conclusion

In this article, we propose a real-time SDRAM controller that bundles read and write commands. We describe the controller architecture and provide a detailed timing analysis of it. We compare our approach analytically and experimentally with other two controllers: the *open-row* one from Wu et al. [6] and the *close-row* AMC [5]. The main results of our evaluation are:

- As the clock speed of SDRAM devices increases, the penalty for data bus turnarounds becomes more significant and, consequently, can limit data bus utilisation. Such challenge is addressed by our controller.
- In scenarios with high operating frequencies, i.e. all investigated modules with the exception of DDR2-800C, the *close-row* controller performs worse than the *open-row* controllers both from the analytical and simulation perspectives.
- Considering only *open-row* controllers, the advantage of our controller over Wu et al. is better highlighted in high-speed modules such as DDR3-1866M (and DDR3-2133N, available in Appendix E).
- Finally, *close-row* controllers consume more power than *open-row* ones. Moreover, when comparing only *open-row* controllers, ours provide a small reduction in power consumption.

ACKNOWLEDGMENTS

This work was partially funded within the EMC2 project by the German Federal Ministry of Education and Research with funding ID 01—S14002O and by the ARTEMIS Joint Undertaking under grant agreement n.° 621429. The responsibility for the content remains with the authors.

REFERENCES

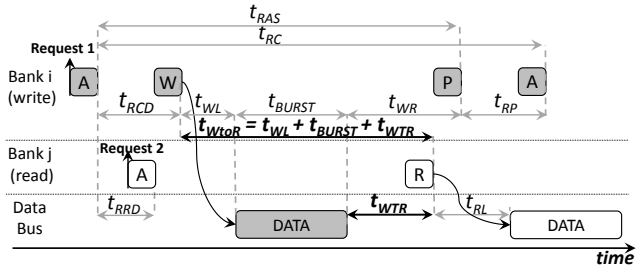
[1] Mellanox Technologies, "Tile-gx36 processor - product brief," 2015-2016. [Online]. Available: http://www. mellanox. com
[2] ——, "Tile-gx72 processor - product brief," 2015-2016. [Online]. Available: http://www. mellanox. com
[3] B. Akesson *et al.*, "Predator: A Predictable SDRAM Memory Controller," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM Press New York, NY, USA, Sep. 2007, pp. 251–256.
[4] J. Reineke *et al.*, "Pret dram controller: bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 99–108.
[5] M. Paolieri *et al.*, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, Dec 2009.
[6] Z. P. Wu *et al.*, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 372–383.
[7] Y. Krishnapillai *et al.*, "A rank-switching, open-row dram controller for time-predictable systems," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 27–38.
[8] B. Akesson *et al.*, "Automatic generation of efficient predictable memory patterns," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, vol. 1, Aug 2011, pp. 177–184.
[9] S. Goossens *et al.*, "Memory-map selection for firm real-time sdram controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 828 –831.
[10] H. Shah *et al.*, "Bounding sdram interference: Detailed analysis vs. latency-rate analysis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 308–313.
[11] Z. P. Wu *et al.*, "A composable worst case latency analysis for multi-rank dram devices under open row policy," *Real-Time Systems*, pp. 1–47, 2016.
[12] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *Real-Time Systems Symposium (RTSS), 2015 IEEE*, Dec 2015, pp. 53–64.
[13] N. Chatterjee *et al.*, "Staged reads: Mitigating the impact of dram writes on dram reads," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, Feb 2012, pp. 1–12.
[14] H. Yun *et al.*, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, July 2015, pp. 184–195.
[15] L. Ecco *et al.*, "Minimizing DRAM rank switching overhead for improved timing bounds and performance," in *Euromicro Conference on Real-Time Systems (ECRTS) 2016*, July 2016.
[16] L. Ecco and R. Ernst, "Technical report: Designing high-performance real-time sdram controllers for many-core systems (revision 1.0)," Braunschweig, 2017.
[17] *JESD79-2F: DDR2 SDRAM Specification*, JEDEC, Arlington, Va, USA, Nov. 2009.
[18] *JESD79-3F: DDR3 SDRAM Specification*, JEDEC, Arlington, Va, USA, Jul. 2012.
[19] *JESD79-4: DDR4 SDRAM Specification*, JEDEC, Arlington, Va, USA, Sep. 2012.
[20] L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 367–376.
[21] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, July 2009.
[22] ARM® Holdings, "ARM Cortex-A Series: Programmer's Guide for ARMv8-A (version 1.0)," March 2015. [Online]. Available: https://static.docs.arm.com/den0024/a/DEN0024.pdf
[23] R. Bourgade *et al.*, "Accurate analysis of memory latencies for wcet estimation," in *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.
[24] H. Kim *et al.*, "Bounding memory interference delay in cots-based multi-core systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, April 2014, pp. 145–154.
[25] B. Bhat and F. Mueller, "Making dram refresh predictable," *Real-Time Systems*, vol. 47, no. 5, pp. 430–453, 2011.
[26] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[27] K. Chandrasekar *et al.*, "Drampower: Open-source dram power and energy estimation tool." [Online]. Available: http://www.drampower.info
[28] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.
[29] Micron Technology, Inc. [Online]. Available: http://www.micron.com

Leonardo Ecco Leonardo Ecco received a bachelor degree in computer science from the Federal University of Santa Catarina, Brazil, in 2007, and a master degree in computer science from the University of Campinas (Unicamp), Brazil, in 2010. He is currently working towards a Ph.D. degree at the Institute of Computer and Network Engineering in the Technische Universitaet Braunschweig, Germany, focusing on real-time SDRAM controllers.

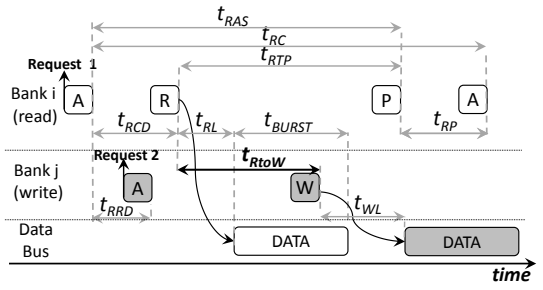
Rolf Ernst Rolf Ernst received a Diploma degree in computer science and the Dr. Ing. degree in electrical engineering from the University of Erlangen-Nuremberg, Erlangen, Germany, in 1981 and 1987, respectively. From 1988 to 1989, he was with Bell Laboratories, Allentown, PA. Since 1990, he has been a professor of electrical engineering with the Technische Universitaet Braunschweig, Braunschweig, Germany. His research activities include embedded system design and design automation.

APPENDIX A
GRAPHICAL DEPICTION OF TIMING CONSTRAINTS

We illustrate the constraints (with the exception of t_{FAW}) in Figs. 16a and 16b. The commands for banks i and j are depicted in different axes simply for clarity. In SDRAMs, there is only one command bus shared by all banks, and it supports at most one command per cycle. Notice that different banks enjoy a certain degree of parallelism, i.e. it is possible to execute a command for bank i while bank j is transferring data.



(a) Write request to bank i followed by read request to bank j.



(b) Read request to bank i followed by write request to bank j.

Fig. 16. SDRAM timing constraints. Write and read requests are depicted in gray and white, respectively. In the figures, the minimum distances between *read* and *write* commands are highlighted.

We illustrate the t_{FAW} constraint in Fig. 17.

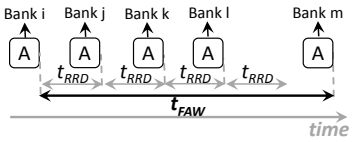


Fig. 17. The t_{FAW} constraint. Notice that $t_{FAW} > 4 \cdot t_{RRD}$.

APPENDIX B
WORST-CASE LATENCY OF WRITE COMMANDS

The worst-case latency of a *write* command is symmetrical to the one of a *read* command. Hence, we simply state Lemma 5 and Theorem 5 and omit proofs.

Lemma 5. *The worst-case latency of a* write *(that is inserted into cr t_{DELAY} cycles after the previous CAS that occupied cr is executed) is calculated with Eq. 22.*

$$L^W(t_{DELAY}) = \max\{t_{prev\text{-}round} - t_{DELAY}, 0\} + t_{curr\text{-}round} \quad (22)$$

where:

$$t_{prev\text{-}round} = (nB - 2) \cdot t_{CCD} + t_{WtoR} \quad (23)$$

$$t_{curr\text{-}round} = (nB - 1) \cdot t_{CCD} + t_{RtoW} \quad (24)$$

Theorem 5. *The worst-case latency of a* write *is given by Eq. 25 if it succeeds a CAS, and by Eq. 26 if it succeeds a non-CAS.*

$$L^W_{SC} = L^W(t_{WL} + t_{BURST}) \quad (25)$$

$$L^W_{SNC} = L^W(t_{WL} + t_{BURST} + t_{RP} + t_{RCD}) \quad (26)$$

Similarly to the case for *read* commands, Theorem 5 relies on the subjective *not-too-late* assumption discussed in Sections 3.3.1 and 4.2. However, an analytical bound independent from such assumption can be computed by simply enforcing that $L^W_{SNC} = L^W_{SC} = L^W(1)$.

APPENDIX C
WORST-CASE LATENCY OF WRITE REQUESTS

The worst-case latency of write requests is symmetrical to the ones of read requests. Hence, we simply state Lemmas 6 and 7 and omit a proofs.

Lemma 6. *The worst-case latency of a WM request is given by Eq. 27.*

$$L^{WM}_{Req} = (t_{Residual} + t_{RP} + t_{RCD}) + (L^P + L^A + L^W_{SNC}) + (t_{WL} + t_{BURST}) \quad (27)$$

Lemma 7. *The worst-case latency of a WH request is given by Eq. 28.*

$$L^{WH}_{Req} = (L^W_{SC}) + (t_{WL} + t_{BURST}) \quad (28)$$

APPENDIX D
THE $\alpha_{PA}(n)$ FUNCTION

As discussed in Lemma 2, we bound the interference that higher-priority CAS commands have in non-CAS commands with the $\alpha_{PA}(n)$ function. For ease of understanding, we provide a graphical depiction of the outcome obtained invoking such function with n=4 in Fig. 18. Notice that, in the figure, we consider a total of nB=5 banks, i.e. nB=n+1, so that the fifth bank can provide higher-priority CAS commands.

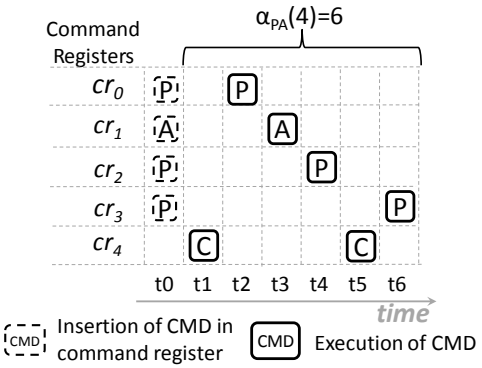
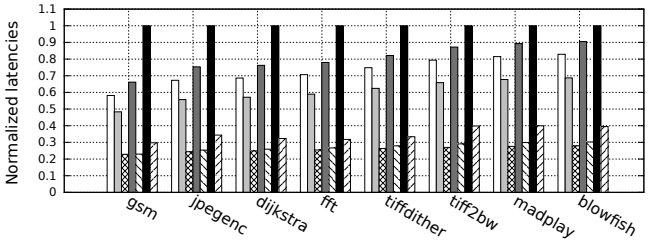


Fig. 18. Example of the outcome of the $\alpha_{PA}(n)$ function in a scenario in which $t_{BURST} = t_{CCD} = 4$ and in which nB=5. In the figure, the letter C refers to a CAS command.

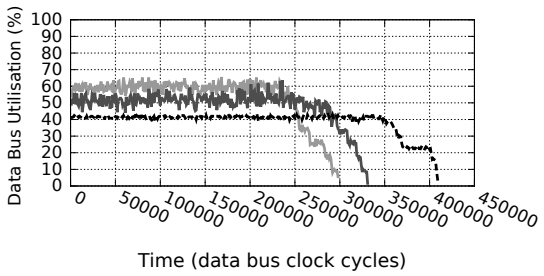
Notice that the sequence of n=4 commands under consideration contains only one *activate* command. This is because if there were for instance two consecutive *activates*, they would be able to postpone each other by more than one cycle (actually by t_{RRD}), a scenario that is not covered by Lemma 2.

APPENDIX E
COMPARISON USING A DDR3-2133N MODULE

We present a comparison of worst-case cumulative SDRAM latencies and data bus utilisation for a DDR3-2133N module in Figs. 19a and 19b, respectively. Notice that we do not provide power data. This is because although DDR3-2133N is described in the DDR3 standard [18], Micron does not manufacture DDR3-2133N devices (and, hence, does not provide a datasheet with their electrical characteristics).



(a) Worst-case (and experimental) cumulative SDRAM latencies. The same color scheme as from Fig. 13 is employed.



(b) Data bus utilisation. The same color scheme as from Fig. 14 is employed.

Fig. 19. Comparison using a DDR3-2133N module.